Random Forest Classification with Hyperparameter Tuning and Handling Imbalanced Data

By – Chakshu Sharma (23081435) Github

Introduction

Hello, and welcome to this tutorial Random Forest Classification with Hyperparameter Tuning and Handling Imbalanced Data.

Machine Learning has been found immensely helpful in a variety of applications, especially in the medical field, where making precise predictions can be a matter of life and death. Among the numerous machine learning algorithms, Random Forest has been a popular and reliable choice. It employs many decision trees to generate accurate, stable, and interpretable predictions. Applying a Random Forest model with default parameters never leads to the best results.

Here in this tutorial, we're going to do it step-by-step, explaining how to construct a Random Forest classifier in the best possible way, make its prediction even better, and solve common real-world problems.

Personally, I myself did not realize the significance of model parameter tuning and class imbalance handling at first. My initial results were encouraging—until I tried to apply the model to new data and discovered that performance wasn't quite as consistent as I'd have hoped. It wasn't until I explored these concepts in more detail that I came to realize their full potential. I learned how to utilize these techniques and how they effectively improved my models, especially when working with life-and-death medical data where validity and accuracy are literally a matter of life and death.

To do this, we'll work through a real dataset—**the Heart Failure Clinical Records dataset**—where we're attempting to forecast patient survival. We'll encounter real-world problems such as imbalanced classes, the requirement for feature scaling, and the selection of hyperparameters with care along the way.

By the end of this tutorial, you'll not only have a stronger understanding of Random Forest but will also know exactly how to approach building models capable of handling practical challenges. Let's dive in!

What Exactly is a Random Forest?

Random Forest is a type of **ensemble learning**, and that means that it uses several simpler models—decision trees in this case—to build one more accurate and resilient model. Use the analogy as follows: suppose you had to make a very important decision, you would ask several knowledgeable individuals what they would do and act on the majority vote or the consensus advice. Random Forest essentially does the same:

- It generates several different decision trees.
- Every tree has a prediction.
- Finally, the Random Forest combines all of these predictions in order to produce the final decision.

You might be thinking, **"Why do we use multiple trees instead of a single one?"** Well, single decision trees by themselves are good but overfit, i.e., learn noise or random variability in training data too well and make poor predictions on new data.

Random Forest tackles this by:

- **Bagging (Bootstrap Aggregating):** Each tree is trained on a slightly different set of data (randomly selected with replacement from the original data set). This ensures that trees see slightly different patterns and therefore their predictions become stronger together.
- **Random Feature Selection:** For each node split in a tree, Random Forest picks a random subset of features rather than testing all of them. This introduces additional randomness, again preventing overfitting.

When I first heard about Random Forest I would wonder, **"How can introducing randomness ever make the predictions better?"** But then after experimenting, I realized this randomness reduces the correlation between trees so that the overall predictions are more stable and accurate.

Advantages of Random Forests:

- **Good Predictions:** Generally, more accurate than an individual decision tree.
- **Noise**: Less sensitive to outliers or irrelevant features.
- **Minimal Parameter Tuning:** Even with default parameters, it generally performs wonderfully (but tuning, as we shall soon see, makes it even better!).
- **Feature Importance:** Provides a good sense of the features that most impact predictions, assisting with interpretability.

Applications in Healthcare (Why it Matters for our Tutorial)

With our application being predicting patient outcomes, these advantages become crucial. Accurate and reliable predictions directly impact clinical decision and patient treatment. This practical application makes it highly gratifying to know and apply Random Forest well.

Let’s begin applying this and address the real problems we face with medical datasets.

Our Dataset: Heart Failure Clinical Records

The data that we will be using in this tutorial is the Heart Failure Clinical Records data, which is an open dataset offering clinical records for patients who've experienced heart failure. It's a fascinating dataset because what we're trying to predict is patient mortality—death or survival—based on the clinical variables collected at hospitalization.

Dataset Overview:

age	age of the patient (years)
anaemia	decrease of red blood cells or hemoglobin (boolean)
creatinine_phosphokinase	level of the CPK enzyme in the blood (mcg/L)
diabetes	if the patient has diabetes (boolean)
ejection_fraction	percentage of blood leaving the heart at each contraction (percentage)
high_blood_pressure	if the patient has hypertension (boolean)

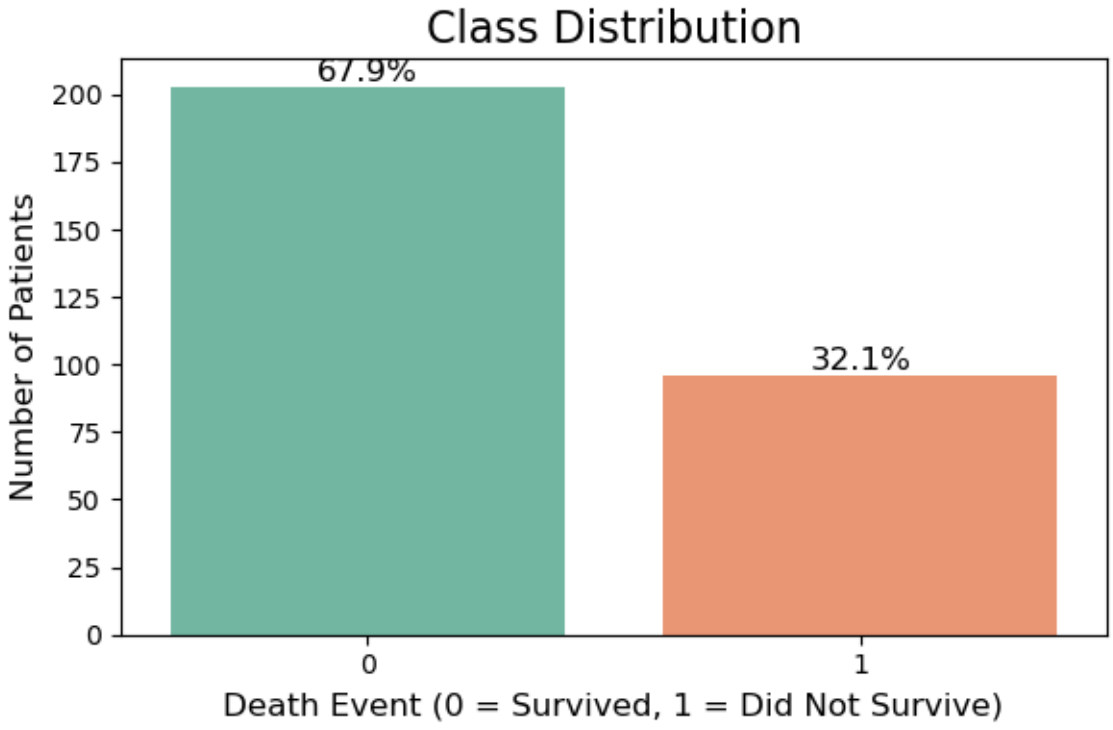
platelets	platelets in the blood (kiloplatelets/mL)
serum_creatinine	level of serum creatinine in the blood (mg/dL)
serum_sodium	level of serum sodium in the blood (mEq/L)
sex	woman or man (binary)
smoking	if the patient smokes or not (boolean)
time	follow-up period (days)
[Target] DEATH_EVENT	if the patient died during the follow-up period (boolean)

Class Imbalance

Medical data is class imbalanced, in which one class dominates the other. In our data:

- Survived (0): ~68%
- Did Not Survive (1): ~32%

Early on, I saw what seemed to be high accuracy but realized the model was only overpredicting survival. In healthcare, failure to detect high-risk patients leads to unwarranted delay in intervention with devastating consequences. Therefore, addressing imbalance was my biggest concern. We'll show how SMOTE (Synthetic Minority Over-sampling Technique) addresses this by creating synthetic samples of the minority class, leading to more reliable predictions.



Feature Scaling and Selection

Though Random Forests are tree-based, scaling does prevent performance from diverging, especially when hyperparameters are also being optimized. We use:

- **PowerTransformer (Yeo-Johnson)** to reduce skewness.
- **MinMaxScaler** to normalize values between 0 and 1.

Along the way, we'll cover each of these in detail, illustrating their use in practice in a natural and straightforward manner.

Random Forest Classifier

Let's start by loading our dataset, splitting it into training and test sets, and training a Random Forest classifier with default parameters. Here's how to do it:

```
# Separate features and target
X = df.drop('DEATH_EVENT', axis=1)
y = df['DEATH_EVENT']

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42
)

rf_simple = RandomForestClassifier(random_state=42)
rf_simple.fit(X_train, y_train)

y_pred_simple = rf_simple.predict(X_test)
print('Random Forest Classification Report')
print(classification_report(y_test, y_pred_simple))
```

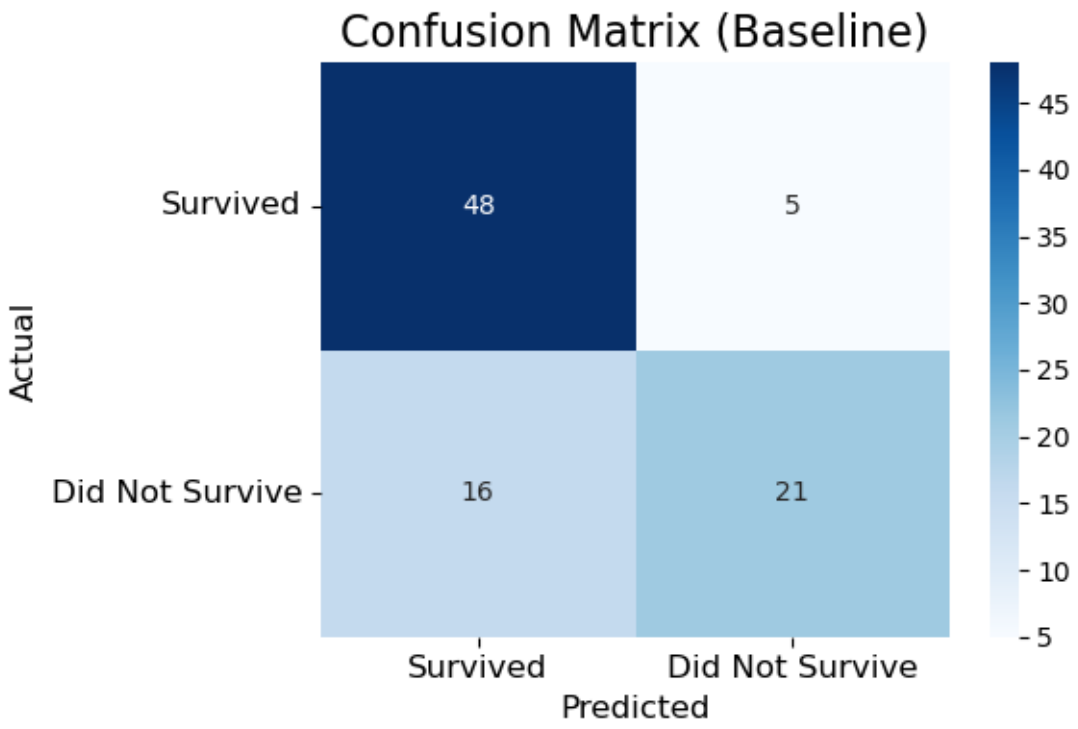
```
Random Forest Classification Report
              precision    recall  f1-score   support

           0       0.75      0.91      0.82        53
           1       0.81      0.57      0.67        37

    accuracy                           0.77        90
   macro avg       0.78      0.74      0.74        90
weighted avg       0.77      0.77      0.76        90
```

When I initially trained the baseline model, the accuracy was 75-80%. At first sight, this seemed promising.

However, a more careful look at measures like recall and precision offered vital information:

- **Recall** (sensitivity to detection of true deceased cases) was considerably worse than what accuracy suggested.
- **Precision** (predicting deceased cases correctly without creating false positives) was good but not great.

One measure like accuracy is deceptive, especially with imbalanced data like ours. Accuracy will be distorted because the model will be predicting the majority class most of the time. Let's visualize this issue by observing the confusion matrix:



Understanding Model Performance Metrics (Accuracy, Recall, Precision)

Before we optimize our model once more, it's crucial to have the different metrics that we use to evaluate machine learning models truly understood. All metrics are not created equal, and what you select will heavily rely on what the nature of your problem is—in our case, a health application.

Let's first define the key metrics briefly:

1. **Accuracy**
   Accuracy determines the number of correct predictions made for cases (survivors as well as non-survivors) out of all predictions. Mathematically, it is:

$$\text{Accuracy} = \frac{\text{Total Predictions}}{\text{Correct Predictions}}$$

   Though accuracy feels natural, it's only useful when your classes are well-balanced. In imbalanced datasets—such as ours—it can be deceptive. For example, classifying all patients as survivors may still give high accuracy because there are a lot of survivors, but it's not useful in practice.
2. **Recall (Sensitivity)**
   Recall, or Sensitivity, indicates how well your model is detecting true positive cases (in our dataset, the patients who did not survive). It is computed as:

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

In health care uses, recall is particularly crucial. Good recall implies fewer positive cases (e.g., at-risk patients at risk of death) are missed. Missing such cases may be life-critical in health care, perhaps delaying necessary medical treatment.

3. **Precision**
   Precision informs you about how well your positive predictions are. That is, of all the patients you predicted "at-risk," how many actually were at risk:

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

   High accuracy in healthcare means fewer false alarms. It reduces unnecessary stress, treatments, or resource allocation on patients inappropriately predicted as at risk.

Initially, I focused on accuracy alone, but soon realized it wasn't enough. The real questions were:

- Did we identify all critical (high-risk) patients? (Recall)
- Out of those we labeled "at-risk," how many truly were? (Precision)

In medical settings, missing someone who urgently needs care (low recall) can be disastrous, while generating constant false alarms (low precision) wastes resources and causes undue stress. Therefore, balancing recall and precision became my central goal.

Balancing Precision and Recall: The F1-Score

To get a balance between recall and precision, we use the F1-score, a balanced metric that takes an average of the two into one number. It provides a better description of your model's general performance in sensitive applications like healthcare:

$$\text{F-1 Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

A high F1-score indicates your model is performing well on both precision and recall, achieving a good balance between both detecting true positives and minimizing false alarms.

Our Goal Going Forward

In our medical setting, optimizing recall (less missing key patients) was my highest priority, and it was very quickly followed by precision. As accuracy remained informative, optimizing recall and precision better captured our goal of reliable, responsible predictions.

In the following sections, you'll see how **class imbalance processing, feature scaling, and hyperparameter optimization** obviously helped us achieve much improved recall and precision.

Enhancing Our Model: Handling Class Imbalance with SMOTE

As we had learned in the earlier step, our baseline Random Forest model was performing very badly in classifying non-surviving patients. The reason was that our data is imbalanced—there were far more surviving patients than deceased patients, and therefore the model was biased towards the majority class.

In order to correct this problem, we shall employ a known method **named SMOTE (Synthetic Minority Over-sampling Technique)**.

What is SMOTE, Precisely?

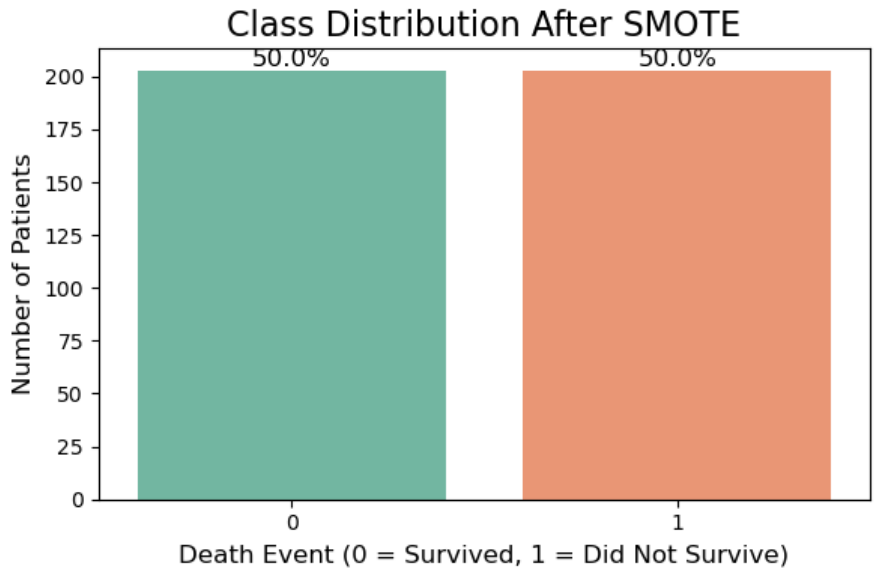
SMOTE overcomes class imbalance by generating artificial (synthetic) samples of the minority class, leveling your data. Unlike the repetition of minority-class records (which would lead to overfitting), SMOTE generates naturalistic artificial instances through feature interpolation from actual minority-class records.

When I first heard of SMOTE, I thought it was complicated. But in fact, it's simple:

- SMOTE uses minority-class instances and identifies their nearest neighbors.
- It generates synthetic data points by interpolating between the neighbors.
- The end result is an even data set without simply duplicating records.

Implementing SMOTE in Our Dataset

```
X = df.drop('DEATH_EVENT', axis=1)
y = df['DEATH_EVENT']

smote = SMOTE(random_state=42)
X_sm, y_sm = smote.fit_resample(X, y)

# Plot balanced classes after SMOTE
plt.figure(figsize=(6,4))
sns.countplot(x=y_smote, palette='Set2')

plt.title('Class Distribution After SMOTE', fontsize=16)
plt.xlabel('Death Event (0 = Survived, 1 = Did Not Survive)', fontsize=12)
plt.ylabel('Number of Patients', fontsize=12)

total = len(y_smote)
for p in plt.gca().patches:
    percentage = f'{100 * p.get_height() / total:.1f}%'
    plt.text(p.get_x() + p.get_width() / 2., p.get_height(), percentage,
             ha='center', va='bottom', fontsize=12)

plt.tight_layout()
plt.show()
```



Now, observe the difference! SMOTE has oversampled our classes to perfection such that both classes—surviving patients and deceased patients—are equally weighted in a dataset.

- **Without SMOTE:** Our model learns only from the majority class (survivors) and misses important information from the minority class (non-surviving patients). This leads to incorrect identification of high-risk patients.
- **With SMOTE:** Our model learns from both classes equally, significantly improving its ability to correctly identify patients at high risk of death, thus enabling timely medical intervention.

The model predictions now would be balanced and predictable. This was an eye-opener, and it became clear to me at the time how important resolving class imbalance truly is.

In the following sections, we shall train our Random Forest classifier from this balanced dataset, once again fine-tuning it by scaling features and carefully tuning hyperparameters.

Feature Scaling and Feature Selection

With the class imbalance successfully addressed using SMOTE, feature optimization is the next key step. While Random Forest models are less sensitive to feature scales than other algorithms, feature scaling and careful selection can still greatly enhance model performance and stability.

Why Feature Scaling?

Feature scaling causes all features to be on similar scales and avoids some features dominating due to their larger numeric ranges. I used to believe that scaling was unnecessary for tree-based models like Random Forests. In practice, however, I've found that applying scaling stabilizes performance, especially under hyperparameter tuning and cross-validation.

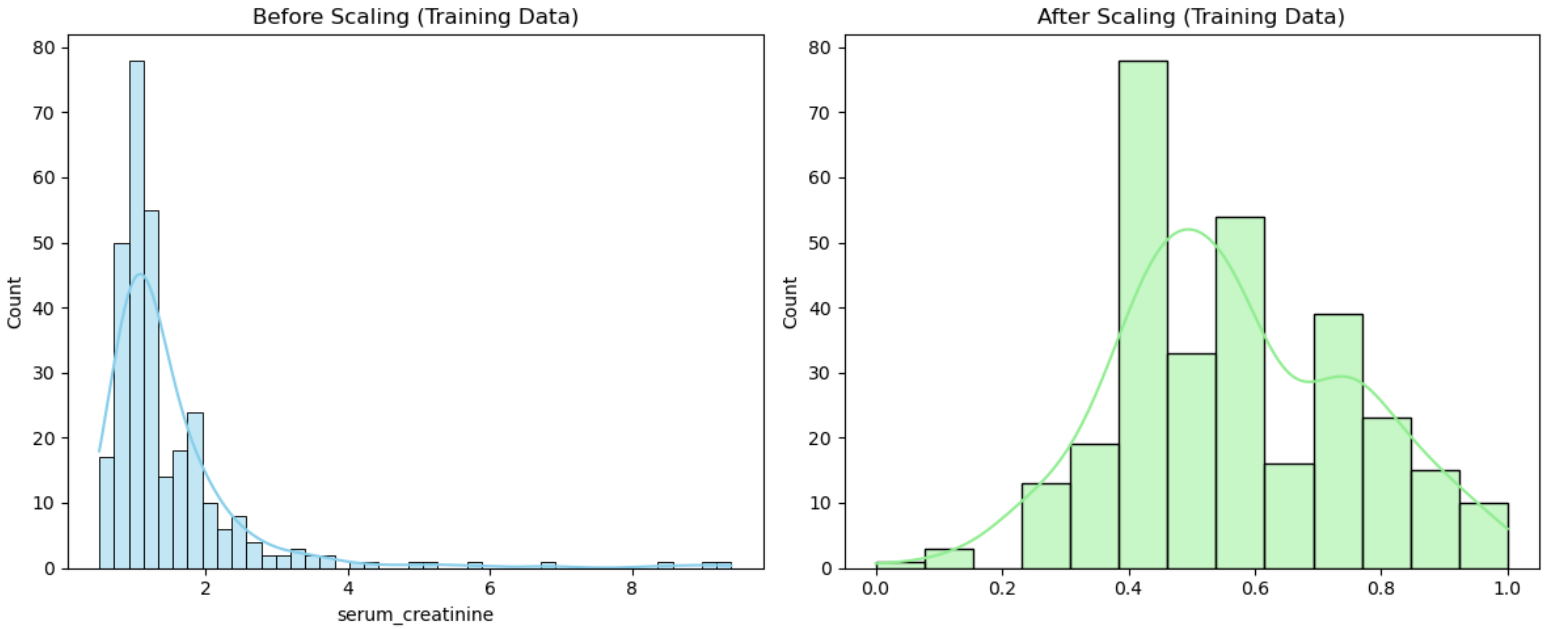
We'll apply two scaling methods explicitly and in order:

- **PowerTransformer (Yeo-Johnson method):** Transforms feature distributions to be more normal-like, counteracting skewness.
- **MinMaxScaler:** Scales the data to a predefined range (0 to 1) again, making it more consistent.

It's **critical** to split your dataset into training and testing subsets **before** scaling. Scaling parameters must be learned **only from training data** to avoid data leakage—a common mistake that initially confused me.

```
X_train, X_test, y_train, y_test = train_test_split(
    X_sm, y_sm, test_size=0.25, random_state=42, stratify=y_sm
)

pt = PowerTransformer(method='yeo-johnson')
X_train_pt = pt.fit_transform(X_train)
X_test_pt = pt.transform(X_test)

mm = MinMaxScaler()
X_train_scaled = mm.fit_transform(X_train_pt)
X_test_scaled = mm.transform(X_test_pt)
```

Let's clearly visualize how scaling affects the distribution of a crucial feature (e.g., serum creatinine):



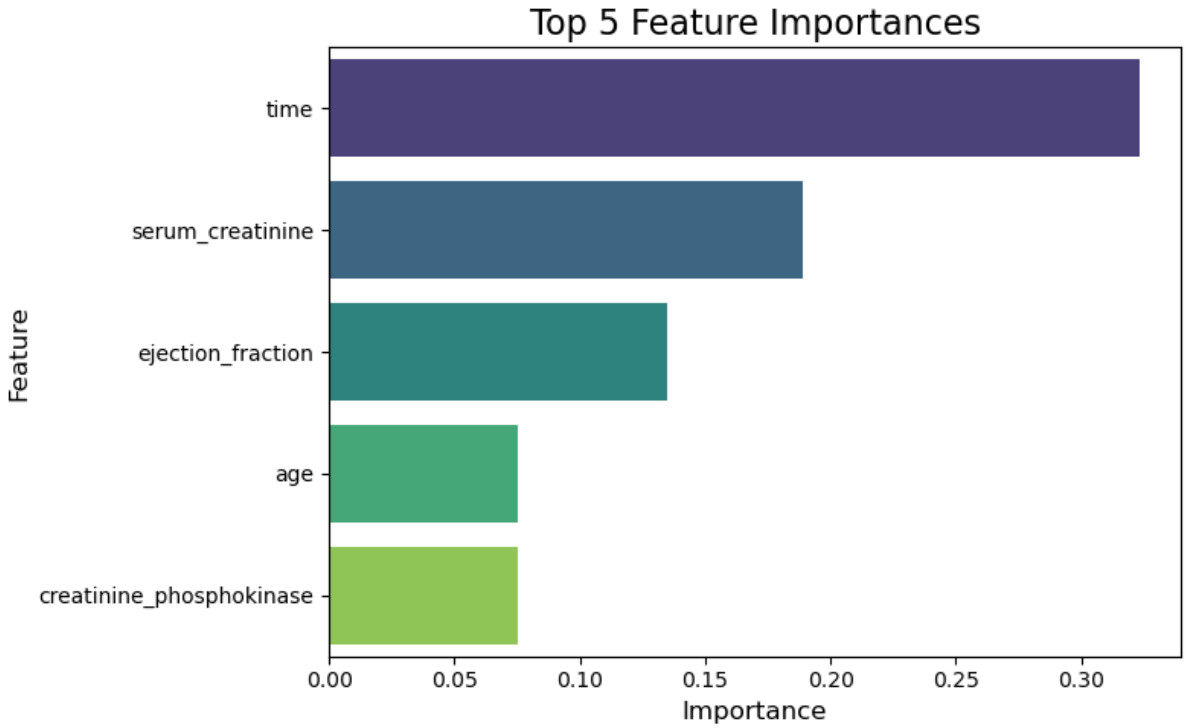
This visualization clearly demonstrates the transformative effect of scaling—reducing skewness and normalizing ranges.

Why Feature Selection?

Feature selection limits your model to the most effective variables, which simplifies the model, reduces noise, and in some cases, makes the model easier to understand. I had previously thought feature selection was all about being efficient with the computations. In fact, however, it helped me uncover critical medical findings by delineating key predictors more clearly.

Let’s implement feature selection and visualise top 5 features in our dataset:

```python
rf_feature = RandomForestClassifier(n_estimators=100, max_depth=10, random_state=42)
rf_feature.fit(X_train_scaled, y_train)

# Extract and sort feature importances
feature_importances = pd.DataFrame({
    'feature': X.columns,
    'importance': rf_feature.feature_importances_
}).sort_values(by='importance', ascending=False)

# Top 5 important features visualization
plt.figure(figsize=(8,5))
sns.barplot(x='importance', y='feature', data=feature_importances.head(5), palette='viridis')

plt.title('Top 5 Feature Importances', fontsize=16)
plt.xlabel('Importance', fontsize=12)
plt.ylabel('Feature', fontsize=12)
plt.tight_layout()
plt.show()
```



This visualization clearly indicates which clinical measurements most strongly influence survival outcomes, providing valuable medical insights.

Hyperparameter Tuning

As we have gone the extra mile to correct class imbalance, normalized features, and selected the most significant variables, our Random Forest classifier is already significantly better. But still, we can go one step further and enhance its performance even more by tuning its hyperparameters.

Why Hyperparameter Tuning?

Hyperparameters are parameters we set before training, and they influence the way your model will learn. I first assumed default hyperparameters would suffice, but tuning them made a huge difference to the predictability accuracy and dependability of the model—most critical in delicate medical applications.

Key Random Forest Hyperparameters:

These are the best hyperparameters we will tweak:

- **n_estimators**: The number of trees within the forest (increasingly more trees usually provides better performance but at a cost of higher training times).
- **max_depth**: Maximum depth of the trees (controls complexity; more depth within trees is at risk of overfitting).
- **min_samples_split**: Minimum amount of samples which must be included within an internal node before being allowed to split (to prevent overfitting).
- **min_samples_leaf**: Minimum amount of samples which must be included within each leaf node (to prevent overfitting).
- **criterion**: Operation used to measure the quality of a split ("gini" or "entropy").

Manually processing combinations of these parameters is not convenient. This is where **RandomizedSearchCV** comes into the picture!

```
rf = RandomForestClassifier(random_state=42, class_weight='balanced')
param_dist = {
    'n_estimators': [50, 100, 150, 200],
    'max_depth': np.arange(2, 20),
    'min_samples_split': [2, 3, 4, 5],
    'min_samples_leaf': [1, 2, 3],
    'criterion': ['gini', 'entropy']
}
search = RandomizedSearchCV(
    rf, param_dist, scoring='f1', n_iter=30, cv=10, random_state=42, n_jobs=-1
)
search.fit(X_train_scaled, y_train)

print('Best hyperparameters:', search.best_params_)
```

```
Best hyperparameters: {'n_estimators': 100, 'min_samples_split': 2, 'min_samples_leaf': 1, 'max_depth': 13, 'criterion': 'gini'}
```

Understanding RandomizedSearchCV

Hyperparameter tuning can be a slow process involving numerous trials and trial-and-error tuning. Fortunately, scikit-learn's RandomizedSearchCV makes it much easier. It checks out a given space of potential hyperparameter values systematically and returns the best fit.

So, what is RandomizedSearchCV?

RandomizedSearchCV randomly tries some number of random hyperparameter combinations from some set or distribution you provide and scores them through cross-validation. As opposed to **GridSearchCV**, which performs an exhaustive search over all possibilities (possibly at huge

computational cost), RandomizedSearchCV searches over a limited number of random possibilities, saving on computation time at the cost of usually achieving the same or very close to the optimal performance.

I was disappointed when I initially heard of hyperparameter tuning. But RandomizedSearchCV made it easy and effective, with satisfactory results without exhaustive searching.

Here is the easy explanation of all the key arguments we used:

- **estimator**: The model to be tuned—in our case, a RandomForestClassifier.
- **param_distributions**: A dictionary of the hyperparameters and their potential values or ranges. RandomizedSearchCV picks randomly from these.
- **scoring**: Defines the metric used to evaluate and optimize for. We've used 'f1' because, in health usage, recall and precision must be traded off against each other.
- **n_iter**: Controls the number of unique random permutations RandomizedSearchCV should try. The larger the value, the more permutations it tests, potentially giving better results but at the cost of longer computation time.
- **cv**: Number of folds in cross-validation to do in order to evaluate each set of combinations. We employ **cv=10** to obtain stable and reliable model performance estimates.
- **random_state**: Ensures reproducibility—defining this enables identical results on several runs.
- **n_jobs**: Allows parallel processing. Employing **n_jobs = -1** utilizes all CPU cores available and significantly speeds up the tuning process.

Tuned Model Results & Evaluation

```
y_pred_advanced = search.predict(X_test_scaled)
print('Random Forest with Tuning')
print(classification_report(y_test, y_pred_advanced))

plt.figure(figsize=(6,4))
sns.heatmap(confusion_matrix(y_test, y_pred_advanced), annot=True, fmt='d', cmap='Greens')

plt.title('Confusion Matrix (Tuned Model)', fontsize=16)
plt.xlabel('Predicted', fontsize=12)
plt.ylabel('Actual', fontsize=12)
plt.xticks([0.5, 1.5], ['Survived', 'Did Not Survive'], fontsize=12)
plt.yticks([0.5, 1.5], ['Survived', 'Did Not Survive'], fontsize=12, rotation=0)
plt.tight_layout()
plt.show()
```

```
Random Forest with Tuning
              precision    recall  f1-score   support

           0       0.92      0.86      0.89        51
           1       0.87      0.92      0.90        51

    accuracy                           0.89       102
   macro avg       0.89      0.89      0.89       102
weighted avg       0.89      0.89      0.89       102
```

These are the final results of our model after fine tuning it. Let's understand it more clearly:

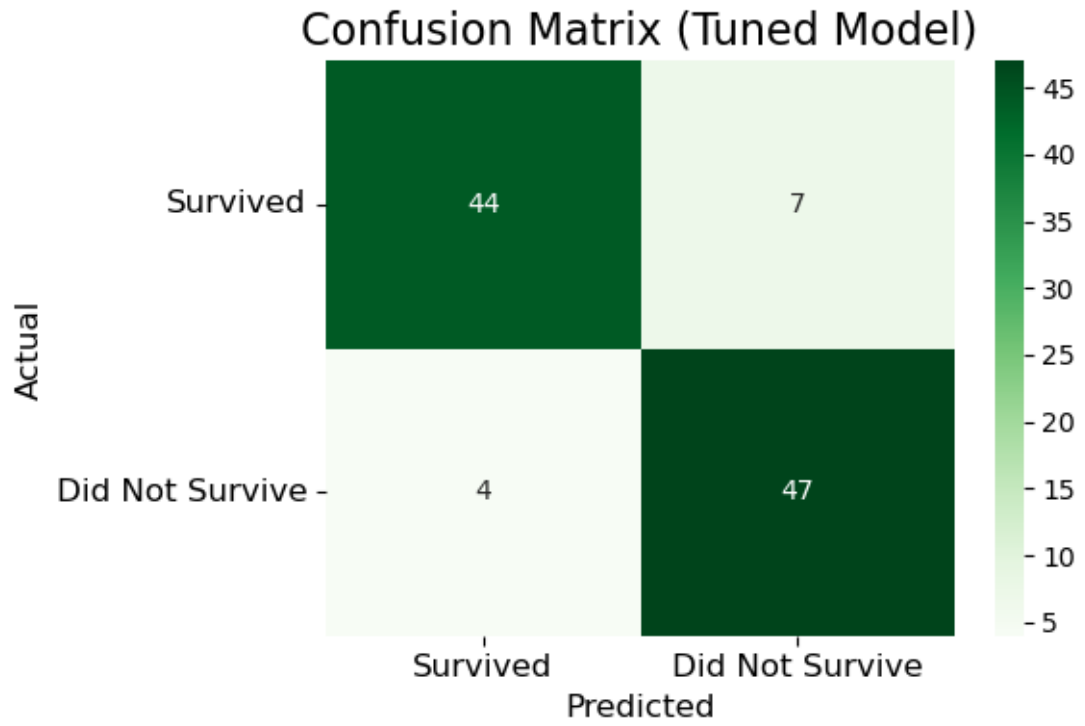
1. **Precision**:

- Precision is 92% for class 0 (Survived) means out of all patients predicted to survive, 92% actually survived.
- Precision is 87% for class 1 (Did Not Survive) means out of all patients predicted to not survive, 87% were correctly identified.

2. **Recall (Sensitivity):**
   - For class 0 (Survived), recall is 86%, or the model correctly identified 86% of actual survivors.
   - For class 1 (Did Not Survive), recall is 92%, or the model correctly identified 92% of actual non-survivors, significantly reducing chances of missing critical patients.
3. **F1-score**: Clearly balances precision and recall with high scores of 89% (Survived) and 90% (Did Not Survive), upholding solid and consistent performance.
4. **Accuracy:** 89% overall accuracy indicates excellent overall prediction performance, both classes balanced.

These improvements demonstrate a significant improvement over our initial baseline model, especially in elimination of major misclassifications.

Confusion Matrix Visualization



- **True Positives (Did Not Survive classified correctly)**: 47 instances correctly predicted as "Did Not Survive."
- **True Negatives (Survivors identified correctly):** 44 instances correctly identified as survivors.
- **False Positives (Incorrectly predicted as Did Not Survive)**: 7 samples incorrectly identified as non-survivors.
- **False Negatives (Incorrectly missed cases of Did Not Survive**): Only 4 non-surviving patients were incorrectly predicted as survivors.

This evident visualisation guarantees that our trained model drastically reduces errors, particularly critical in healthcare contexts where precise predictions of patient outcomes can directly influence medical interventions.

They strongly validate the systematic improvements we made, including **SMOTE**, **scaling**, **feature selection**, and **hyperparameter optimization**, and their real-world efficacy.

Conclusion

We have just gone through this rather intensive tour of fine-tuning a Random Forest classifier with thorough hyperparameter tuning, class balance handling, scaling, and feature selection. A quick revisit of what we learned and covered is due here.

- **Random Forest Fundamentals**: Being in full understanding of the number of trees together making strong predictions.
- **Significance of Metrics**: Not only accuracy, keep in mind—precision and recall are extremely significant in medical cases.
- **Class Imbalance Handling (SMOTE):** Strongly demonstrated that balanced sets greatly improve prediction of critical outcomes.
- **Feature Scaling and Selection**: Strongly demonstrated how these processes stabilize and shrink models, both enhancing performance and interpretability.
- **Hyperparameter Tuning (RandomizedSearchCV):** Strongly demonstrated how structured tuning greatly enhances model reliability and prediction capability.

At first glance, baseline performances were adequate but a closer look easily showed weaknesses. Not all of the approaches used were strictly theoretical—it considerably boosted the model's predictive ability for accurate prediction of patient outcomes, particularly for larger minority classes.

In medicine, accurate predictions literally are life and death in terms of enhanced clinical decision-making and patient treatment, that is why we need to look beyond just accuracy while training a predictive model.

References

- **[Data Source] UCI Machine Learning Repository, Heart Failure Clinical Records Dataset,** https://archive.ics.uci.edu/dataset/519/heart+failure+clinical+records, **Accessed: March 27, 2025.**
- Leo Breiman, "Random Forests," Machine Learning, 2001. https://link.springer.com/article/10.1023/A:1010933404324, Accessed: March 27, 2025.
- Will Koehrsen, "Hyperparameter Tuning the Random Forest in Python," Medium. https://towardsdatascience.com/hyperparameter-tuning-the-random-forest-in-python-using-scikit-learn-28d2aa77dd74, Accessed: March 27, 2025.
- Jason Brownlee, "SMOTE for Imbalanced Classification with Python," Machine Learning Mastery. https://machinelearningmastery.com/smote-oversampling-for-imbalanced-classification/, Accessed: March 27, 2025.